Soft timing closure for soft programmable logic cores: The ARGen approach

Théotime Bollengier, Loïc Lagadec, Mohamad Najem, Jean-Christophe Le Lann, Pierre Guilloux

Soft timing closure for soft programmable logic cores: The ARGen approach

Théotime Bollengier[1,2], Loïc Lagadec[2], Mohamad Najem[2], Jean-Christophe Le Lann[2], and Pierre Guilloux[3]

[1] B-Com, France
[2] Lab-STICC UMR 6285, France
[3] IRISA UMR 6074, France
loic.lagadec@ensta-bretagne.fr

Abstract. Reconfigurable cores support post-release updates which shortens time-to-market while extending circuits' lifespan. Reconfigurable cores can be provided as hard cores (ASIC) or soft cores (RTL). Soft reconfigurable cores outperform hard reconfigurable cores by preserving the ASIC synthesis flow, at the cost of lowering scalability but also exacerbating timing closure issues. This article tackles these two issues and introduces the ARGen generator that produces scalable soft reconfigurable cores. The architectural template relies on injecting flip-flops into the interconnect, to favor easy and accurate timing estimation. The cores are compliant with the academic standard for place and route environment, making ARGen a one stop shopping point for whoever needs exploitable soft reconfigurable cores.

1 Introduction

As integrated circuits become increasingly complex and expensive to develop, the ability to apply post-fabrication changes appears all the more attractive. A direct gain lies in eliminating the cost and time associated with re-spinning silicon when fixing a bug or specializing the device to a specific application. Embedding reconfigurable logic in designs offers a solution to the semiconductor designers who need to update silicon post production.

In this context, several embedded FPGAs (eFPGA) have been developed as reported in [1] [2] [3]. EFPGAs are flexible logic fabrics, that, once programmed, implement digital circuits. But, unlike FPGAs, eFPGAs are intended to serve as pieces of a whole system-on-chip design. This approach allows:

- To support easy design specialization, while promoting reuse among several applications,
- To fix design issues that would have been belatedly detected (only after fabrication, if not post delivery),
- To add on-demand fleeting functionalities, such as assertion-based monitoring[4].
- To reflect changes in design specifications. This shortens time-to-market by allowing starting the design ahead of full specification availability (eg. to

support changes in an evolving standard). The case of the H.264/AVC standard - that includes 22 revisions, corrigenda, and amendments spanning from May 2003 to February 2014 - helps assessing how serious this issue is.

This obviously comes at the cost of area and performance overheads, compared to a straight silicon implementation. However there are even more serious limitations[5].

First, every eFPGA embeds a fixed amount of reconfigurable resources. Any mismatch between theses resources and the applications needs (in terms of amount and nature of resources) is a serious issue. It may either prevent from using this support (if the application requirements exceed the eFPGA resources) or lead to a poor resources usage (internal fragmentation may nullify the advantage of using an optimized hard eFPGA core). Then, tailoring eFPGAs in order to set up a product line may seem attractive. Unfortunately, customizing eFPGA size and resources towards an application domain is likely to cause lengthy development cycles, as each new instance of hard eFPGA core must be silicon proven. However, Kuon et al. [6] demonstrated automation of circuit design, layout and verification, to cut off the required effort and time to design a new embedded hard FPGA core.

Second, eFPGAs are hard IP cores, which integration is complex and time consuming, and raises technology compliance issues, as all the cores must be provided with the same technology. As an example, the System-on-Chip of [7] was a scalable system infrastructure hosting heterogeneous reconfigurable accelerators, whose implementation required to migrate one of the accelerators to 90-nm, which resulted in a 6 months extra work.

This incites to move up a level of abstraction, based on soft macros that are process-independent. Some works have been reported in designing Soft Programmable Logic Cores (SPLCs) as summarized in section 2. This paper complements these previous works by addressing some known issues in terms of scalability and timing closure.

The main contribution is ARGen, a generator of soft reconfigurable cores. ARGen supports core customization and trades a minor overhead against accurate timing closure. Also, SPLCs come along with their programming environment. As a result, the SPLCs' strengths (flexibility, just-fit dimensioning, performances predictability) outweigh disadvantages in term of performances.

The remainder of this paper is organized as follows: section 2 summarizes related work on soft reconfigurable logic cores, section 3 describes the structure of the proposed SPLC, aiming to simplify both SPLC synthesis and system integration, section 4 presents the exploitation tool flow and circuit timing analysis, before section 5.1 reports some results.

2 Background

Soft programmable logic cores (SPLC) have been introduced in [8] [5] to emphasize flexibility and shorten development time, hence promote agility. Unlike hard

core eFPGAs, synthesizable SPLCs are delivered as RTL descriptions, and synthesizing such cores is done using usual tools (standard ASIC or FPGA flows).

Integrating SPLCs in a design is easy : a flat synthesis of designs with one or many SPLCs requires no floorplanning.

Integrating SPLCs is safe : a whole design that contains SPLCs, can be verified, simulated and emulated without additional complexity.

Integrating SPLCs is a just-fit process : SPLCs can be easily customized at the sole cost of updating the RTL description, with no need to silicon-proof each modified instance again, so that domain space exploration may be affordable.

Integrating SPLCs is reversible : the decision to use either a SPLC or fixed logic to implement any subpart of a design remains reversible until just before the chip goes to foundry. This decision stays on the designer who best knows which subsystem may/will need later modifications, and how much flexibility makes sense.

Integrating SPLCs supports optimization : authors in [9] demonstrated that soft core area overhead can be reduced by 58% and the delay overhead by 40% by creating custom standard cells (referred as *tactical cells*) that are more suitable for reconfigurable architecture implementations, and by using a tile-based approach to structure the layout of the hard macro.

As summarized above, SPLCs exhibit valuable features thanks to their RTL nature, nevertheless two difficulties emerge, that prevent from a wide broad adoption. First, the timing paths to explore are many. Second, the awareness of physical timings is poor.

Unlike regular designs, SPLCs present unusually large number of potential timing paths and combinatorial loops, due to their reconfigurable nature. This stresses the synthesis tool and may limit the size and nature of SPLCs [9]. To address this problem, authors in [5] propose to simplify the SPLC architecture by removing programmable flip-flops and by allowing the signal flow to go only in one direction, thus preventing combinatorial loops. As a consequence, the SPLCs exclusively target combinatorial applications; the proposed architecture is minimal which restricts the complexity and nature of applicative circuits to be implemented.

Moreover, performing timing analysis of a circuit mapped on a SPLC may be subject to a physical timing information miss. Exploiting the SPLCs goes through synthesizing applications on the reconfigurable cores. This relies on a synthesis tool -further referred as *virtual* synthesis tool- that is independent from the *physical* synthesis tool (the standard ASIC tool flow) used to implement the SPLC itself. As an example, in [8] [5], the *virtual* synthesis tool is VPR [10]. The *virtual* synthesis tool executes timing-driven placement and routing, as well as timing analysis. These steps require the tool to be aware of every physical delay of SPLC resources. In [8], these physical delays are approximated using the conceptual representation of the SPLC. This results in an inaccurate circuit timing analysis, as adjacent resources in the conceptual SPLC representation may actually be positioned far apart in the silicon, thus tampering the delay estimation.

In [5] and [9], timing exceptions are set to ignore the unused SPLC paths in the mapped circuit netlists when performing timing analysis according to the physical ASIC tools. This ensures the delay measures of the mapped circuits' critical paths are more reliable. However this comes at the cost of back and forth navigation between virtual and physical synthesis tools. Another option would be to extract an accurate information from the physical synthesis to feed the virtual tools. Yet, extracting this information means collecting the elementary delays of all arbitrary sub-segments of all combinatorial paths. This is of high complexity and must be processed for each new SPLC physical synthesis. Besides, this can only be considered a preliminary step, before the virtual synthesis tool actually exploits this information. As a consequence, even if back annotating the SPLC conceptual representation (used by the virtual tool flow) with actual physical delays is considered in [5] [11], it has never been implemented in practice.

Our contribution goes one step beyond, and lifts these limitations. In this work, we propose a template for modifying SPLC architectures. This allows as easy SPLC integration as reported in [5] - but with no restriction on the SPLC architectures - while providing easy and accurate timing analysis of mapped circuits, solely using the virtual synthesis tool.

3 SPLC design

Using an SPLC assumes three pre-requisite steps: generating the SPLC architecture, synthesizing this architecture to a physical target, and supporting system integration. Once generated, the SPLC module becomes a library element that can be instantiated within the application's RTL description, then the whole design is synthesized using an ASIC flow. The portable RTL description of the SPLC supports flat synthesis of the whole design without the need for specific steps such as floorplaning.

Then synthesizing and deploying applications onto the SPLC involve a dedicated software environment. This tool is independent from the physical technology, which in turn may require specific software development, as detailed in section 4.

Fig. 1 shows how these two flows, which together contribute to making SPLC a credible solution, relate and interact. The "ArGen" tool covers two aspects as detailled in the next section: architecture generation and bitstream production.

3.1 Overview of the SPLC architecture

A SPLC architecture is composed of two layers:

- The *computation layer*, which is the set of reconfigurable elements that are available to applications, such as routing wires and function units.
- The *configuration layer*, which configures the computation layer.

The "ARGen" tool reads a specification of the computation layer, to automatically generate the SPLC's RTL description. This specification expresses the

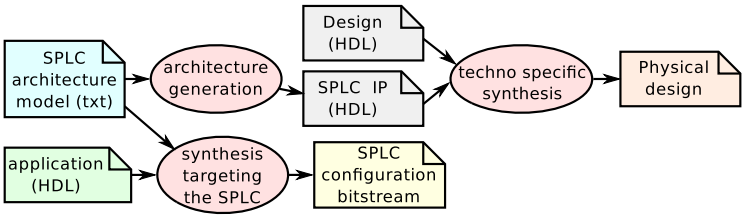

Fig. 1. Complete synthesis flow for using an SPLC.

computation layer resources and their interconnections; the configuration layer is then automatically derived and eventually added to the model. Finally, a model transformation generates VHDL textual description of the architecture , allowing the SPLC module to be instantiated from a user design. The SPLC entity contains clock inputs, a vector of inputs and a vector of outputs, as well as a configuration interface. The generated RTL code is portable, simulation friendly, and synthesizable.

3.2 Detailed Computation Layer

The target SPLC computation layer that serves as a case study for this paper allows the synthesis of a large spectrum of applications. It is a fine-grained generic LUT-based architecture compatible with the standard architectures used in the academic Versatile Place and Route (VPR) tool [10]. This architecture is a simple *island-style* architecture as shown in Fig. 2, composed of Configurable Logic Blocks (CLBs) surrounded by routing channels.

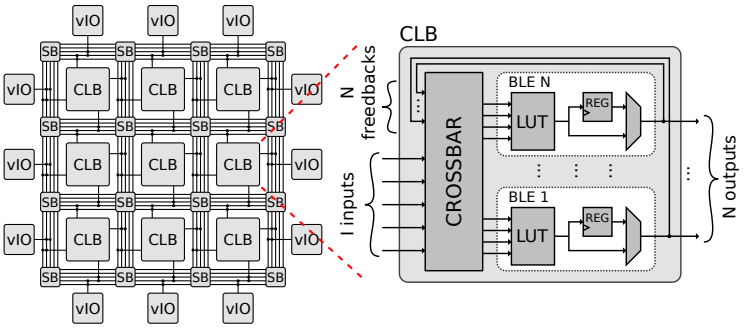

Fig. 2. An illustration of the proposed computation layer with 3×3 CLBs.

The SPLC has $Width \times Height$ CLBs, each of which has I inputs and N outputs. A CLB is composed of N BLEs (Basic Logic Element). A BLE has one LUT with K inputs and one register that can be bypassed (the *application*

register). Inputs of BLEs are derived from a global crossbar with $I + N$ inputs (the I CLB inputs plus N feedback signals from the BLEs outputs). Each routing channel contains W unidirectional wires, in both directions, that can be connected to other wires from adjacent routing channels, depending on how the Switch Blocks (SB) are configured. Connections are implemented as multiplexers that are controlled through their *select signal*(s) coming from the configuration layer (as illustrated in Fig. 3).

The ARGen approach isolates the SPLC conceptual representation from its physical implementation on silicon. The proposed solution is to inject extra registers within the SPLC to latch the output of every configurable multiplexer that connects routing wire tracks. These registers are referred to as *Virtual Time Propagation Registers* (VTPRs). VTPRs break down physical logic chains into short segments, and prevent any combinatorial loop from appearing on the physical SPLC implementation, whichever its configuration. VTPRs are transparent for circuits mapped on the SPLC, and do not appear in the SPLC conceptual model.

VTPRs exhibit two decisive advantages. First, using VTPRs in a SPLC architecture alleviates the task of the physical synthesizer, as VTPRs reduce timing paths in the SPLC architecture and prevent combinatorial loops. This promotes architectures' scalability. There is no more need to limit size and complexity of synthesized architectures, nor to restrict the signal flow in one direction. This, however, rises the need for an extra and faster clock (Clk_{VTPR}), to allow signal propagation through VTPRs within one applicative clock cycle. Second, VTPRs favors timing closure, as reported in section 4.2. VTPRs brings no improvement in term of performances of the synthesized SPLC. In that, VTPRs differ from C-slowing [12] which can be combined with retiming for sake of throughput increase.

3.3 Detailed Configuration Layer

The SPLC configuration is a contiguous sequence of bits that corresponds to the adequate configuration of SPLC resources (LUTs content, Crossbar, CLBs, and SBs) to implement a given application. The configuration layer is implemented as one or multiple shift registers. Once the transfer of the SPLC configuration to this register completes, every bit in the configuration layer is set to the desired value, resulting in the implementation of the synthesized circuit.

3.4 System integration

When a design requires reconfigurability, the designer first isolates the part of the design which is subject to change apart from the static design, thus identifying the signals at the interface. The RTL of the static part instantiates the SPLC module, and connects the interface signals to the SPLC virtual inputs/outputs. A configuration controller drives the SPLC configuration interface, made of an input `config_in` vector and an input `config_valid` bit. The number of configuration shift registers, forcing the size of the `config_in` vector, is determined

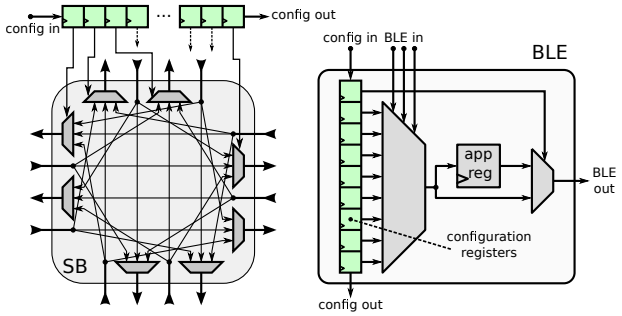

Fig. 3. Implementation of a Switch Box (left) and a BLE (right), whith their associated configuration registers from the configuration layer.

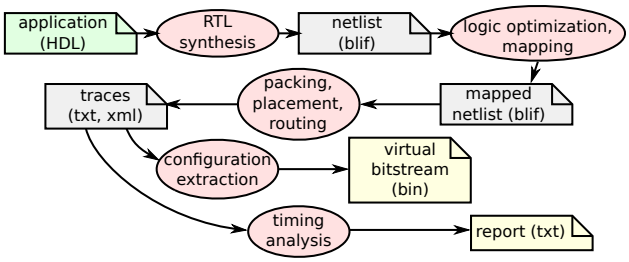

Fig. 4. Synthesis flow: from application RTL to an SPLC configuration

to fit designer's needs (the wider the interface, the faster the configuration, the more area it consumes). The configuration controller can read the SPLC configuration bitstream from an internal memory, be mapped on a bus in case of a SoC, or even be accessible from outside the chip through the pinout.

4 SPLC exploitation

When deploying applications onto SPLCs, no commercial tool fits the architecture, but some open-source academic works have been reported that offer a customizable solution for application synthesis. The ARGen approach relies on existing third parties tools, while offering a fast and accurate timing closure as a strong contribution. To this end, in addition to RTL code, the ARGen tool also generates VPR specific architecture description files. Additionally, ARGen generates bitstream and executes timing analysis.

4.1 Application synthesis targeting the SPLC architecture

Hardware applications are designed as a RTL description within a Hardware Description Language (HDL). First, this description is used to produce a low-level netlist. Any tool can be supported as long as it outputs BLIF format[13] (e.g. the Odin II open-source CAD tool [14] that takes verilog as input). Then

techno-mapping and optimizations take place. In this flow, ABC [15] is used to perform logic synthesis and optimizations, then map the netlist to the SPLC LUTs and produce a new netlist. Then, this netlist can be packed, placed and routed using VPR [10]. Finally, a timing report is generated along with the SPLC configuration bitstream by parsing VPR's outputs, and computing a portion of configuration per each SPLC resource. The synthesis flow of applications for the overlay is summarized in Fig. 4. This flow targets the SPLC architecture at various stages:

- LUT mapping respects the maximum number K of inputs per LUT;
- BLE packing into CLB respects N, the number of BLEs per CLB, as well as I, the number of inputs per CLB;
- Placement exploits the location of SPLC resources;
- Routing conforms to the SPLC routing graph;
- Configuration is compliant with the SPLC bitstream template;
- Timing analysis is specific to the SPLC architecture, as highlighted in the next section.

4.2 Timing analysis

When synthesizing a SPLC, the timing reports indicate the F_{max} frequency at which the design may operate. F_{max} depends on the worst case propagation delay of SPLC atomic resources isolated between two VTPRs.

The virtual synthesis flow only relies on F_{max} to perform timing analysis. At the netlist level, assuming a net N_C connects two logic nodes L_A and L_B, the delay of the mapped net N_C can be computed as the number of VTPRs along the mapped path from L_A to L_B.

Adding VTPRs requires to operate two clocks: Clk_{VTPR}, the VTPRs clock, and Clk_{app}, clocking the application registers. To ensure that the mapped circuit properly runs on the SPLC, Clk_{VTPR} and Clk_{app} must abide by the relation:

$$F_{max} \geq F_{Clk_{VTPR}} \geq N_{VTPR} \times F_{Clk_{app}} \tag{1}$$

where

$$N_{VTPR} = \max_{\forall\ hypernet\ N_c} \left(\max_{N_c^i \in subnets(N_c)} (length(N_c^i)) \right) \tag{2}$$

In eq. 2, the netlist is seen as a set of hypernets. These multi-terminal nets are spread as a collection of monoterminal nets, each of which goes from and reaches either an IO or a register.

This greatly simplifies and speeds up timing computation. Especially as the Manhattan distance is a smart approximation of *length*. Then eq. 2 profitably replaces Elmore delay computation [16].

5 Experiments

The experiments rely on exploring the implementation cost of a parametric SPLC. Then, the use of this SPLC is demonstrated on a regular expression matching application.

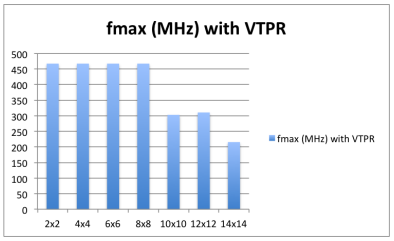

Fig. 5. VTPRs make timings predictable and lead to acceptable frequency

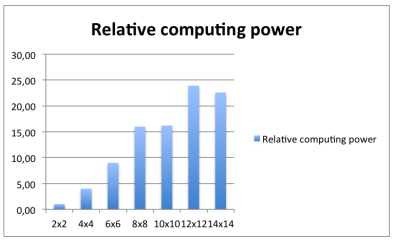

Fig. 6. Computing power exhibits scalability

5.1 SPLC definition

The SPLC structure conforms to the previous specification, with dimensions ranging from $2 * 2$ to $14 * 14$ CLBs, with 4 BLEs per CLB (16 to 764 BLEs). The SPLC is synthesized using the J-2014-09-SP7 version of Synopys Design Compiler, on a ST 65nm technology.

Figures 5, 6, and 7 illustrate some benefits of using VTPRs. Figure 5 and 6 illustrate the max frequency of the SPLC, regarding its dimensions, and the offered computing power respectively. What makes sense to be noticed here is first the top frequency (467 MHz) but also that the computing power ($\#BLE \times f_{max}$) exhibits scalability.

The two following figures illustrate the feasibility of the approach.

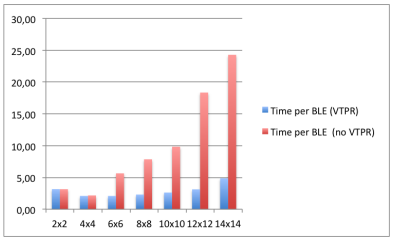

Fig. 7. VTPRs make the synthesis time affordable, hence promote scalability

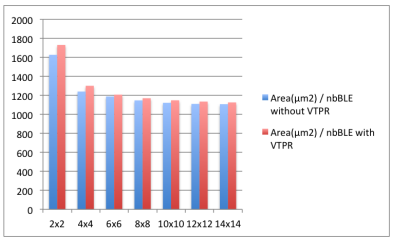

Fig. 8. VTPRs lead to a 3% average overhead in term of area

Figure 7 shows that the synthesis time ranges from 2.08 to 4.85 s per BLE, with 2.89s/BLE in average when using VTPRs. Instead, this average rises up to 10.18 without VTPRs (ranging from 2.17 to 24.28). Besides, the standard deviation is reduced from 8.22 to 0.97 when introducing VTPRs. The lessons learned are two: first VTPRs save synthesis time, second VTPRs make synthesis time predictable. Figure 8 shows that VTPRs come almost for free in term of area (around 2% for bigger SPLCs). Also, as virtual prototyping usualy relies on FPGAs as an experimental platform, Table 1 reports results when implementing SPLCs -with and without VTPRs- on top of Xilinx FPGAs. Three stages are

Table 1. Synthesis time on Xilinx FPGA, with and without VTPR

Dimensions		XST time			MAP time			PAR time			TRCE time			Total Synthesis time		
Size	BLEs	Raw	VTPRs	Ratio	Raw	VTPRs	Ratio	Raw	VTPRs	Ratio	Raw	VTPRs	Ratio	Raw	VTPRs	Ratio
2×2	16	32	32	1.00	133	120	1.11	103	66	1.56	35	33	1.06	303	251	1.21
4×4	64	68	61	1.11	228	238	0.96	344	138	2.49	71	39	1.82	711	476	1.49
6×6	144	159	146	1.09	530	298	1.78	60538	155	390.57	219	49	4.47	61446	648	94.82
8×8	256	328	354	0.93	909	524	1.73	3088	220	14.04	493	63	7.82	4818	1161	4.15
10×10	400	661	715	0.92	1842	763	2.41	4940	350	14.11	1208	84	14.38	8651	1912	4.52
12×12	576	1296	1371	0.94	4838	1179	4.10	39856	474	84.08	3289	105	31.32	49279	3129	15.75
14×14	784	2343	2487	0.94	4613	3904	1.18	18617	654	28.47	5007	154	32.51	30580	7199	4.24

reported: XST (RTL synthesizer), MAP and PAR (logic synthesizer, placer and router), and TRCE (timing analyser).

VTPRs do not significantly impact synthesis time. On the opposite, MAP and PAR show unpredictable execution time unless VTPRs are used. This comes from the heuristics within these tools. In particular, the combinational loops within the SPLCs are broken down into smaller netlists undeterministicaly. TRCE seems to scale with regards to $\#BLEs$. Again, the synthesis time is shorter and more predictable when using VTPRs, which preserves the FPGAs as a potential virtual prototyping platform when designing VTPR aware SPLCs.

5.2 Usage

Embedding a SPLC in a design adds some flexibility, which makes sense in various cases. First, this feature helps designers to fix bugs encountered in the design by offering post release Engineering Change Order (ECO) opportunities. Second, the SPLC can be used to implement transient functions. As an example, hardware probes and monitors may be useful when validating the design, although they are usually removed in a production phase. Last, SPLCs support incorporating new functions while updating some others. It is usually an iterative process to make a design change successfully, and SPLCs naturally support incremental compilation.

This paper focuses on the third item, and promotes the use of SPLC as a support for automata implementation. We consider a regex (regular expression) engine that generates logic to be implemented on a SPLC. The hardware template assumes an initial memory continuously streams data (one byte per cycle) to the generated design whose role is to detect a match with a reference pattern. The detection scheme relies on a non-deterministic finite automata (NFA) [17] to alleviate the need for backtracking (due to its multiple active states). Table 2 illustrates the implementation cost of representative expressions in terms of flip-flops and LUTs in the SPLC. The number of flip-flops only depends on the pattern size, while the number of LUTS does on the pattern complexity. The first five expressions score the cost of |, ?, + and $*$ constructs. The last two illustrate real cases. The *link* expression looks for hyperlinks with a known root. The full expression is: `/<a\s+href="/courses/[^"]*"[^>]*>/`. *ssh* is of higher complexity and corresponds to searching ssh traces in a log file. The full expression

Table 2. Synthesis results

regex	SPLC FF	SPLC LUT	SPLC BLE	min N_{VTPR}	min size	W min
/abcdefgh/	8	12	12	10	2x2	4
/abcd|efgh/	8	15	15	12	2x2	4
/a(bcdefg)?h/	8	13	13	12	2x2	4
/a(bcdefg)+h/	8	14	14	10	2x2	8
/a(bcdefg)*h/	8	16	16	10	2x2	6
link	23	44	44	14	4x4	12
ssh	76	99	100	18	6x6	12

is: `/[^ ]+ +\d+ \d+:\d+:\d+ [^ ]+ sshd\[\d+\]: Accepted (password | publickey) for [^ ]+ from \d+\.\d+\.\d+\.\d+ port \d+ ssh/`

The interesting point is that these expressions can be synthesized on modest SPLCs (6^{th} column in Table 2), quickly enough (1 to 10 seconds, depending on the expression) to support design space exploration. Then, the circuit designer can dimension the SPLC in a just fit approach (last two columns) for a class of regex. The performances are only slightly impacted by the complexity of the expressions. N_{VTPR} denotes the factor by which the clock is divided due to the presence of VTPRs in the routing to generate the applicative clock Clk_{app}. The worst case still exhibits over 25 MHz $F_{Clk_{app}}$ applicative frequency.

6 Conclusion

The decision to include a reconfigurable IP in a design shortens time-to-market by allowing starting early development cycle before full availability of final applicative specifications. The design remains flexible, and the designers can partially update the circuit, even after silicon release. Integrating some Soft Programmable Logic Cores (SPLCs) is the easiest way to gain this flexibility, without affecting the ASIC design flow. However, timing analysis of circuits running on SPLCs usually comes to be inaccurate.

Our contribution tackles this issue by providing SPLCs decorated with VTPRs. VTPRs are extra registers, which break down loops in the interconnect in order to master the timings in the SPLC. This offers simplified timing closure (predictable and accurate timings). Besides, VTPRs ensure scalability when synthesizing the SPLC. Also, VTPRs make sense as an affordable feature, and come at the sole cost of 3% area overhead in average.

Finally, this approach has been demonstrated through implementing regex detection. This use case illustrates how SPLCs can support changing protocols. This work also closely relates to overlays, which are usualy virtual coarse-grain architectures, overlaying on top of fine-grained FPGA devices, for sake of improved productivity, portability, debugging capabilities, etc. ARGEN has demonstrated to suit designer's needs when adressing overlays. Future work will investigate how combining SPLC and overlays can drive new improvements.

Credits

This work has been supported by the French National Research Agency under the contracts ANR-11-INSE-015 (ARDyT) and ANR-A0-AIRT-07. (B-Com)

References

1. Menta - embedded Programmable Logic. http://www.menta-efpga.com.
2. Nanoxplore. http://www.nanoxplore.com.
3. ADICSYS - eFPGA (embedded FPGA) IP. http://www.adicsys.com.
4. M. Abramovici, P. Bradley, K. N. Dwarakanath, P. Levin, G. Memmi, and D. Miller, in *Proceedings of DAC 2006*, E. Sentovich, Ed. ACM, pp. 7–12.
5. S. J. Wilton, N. Kafafi, J. C. Wu, K. A. Bozman, V. O. Aken'Ova, and R. Saleh, "Design considerations for soft embedded programmable logic cores," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 2, pp. 485–497, 2005.
6. I. Kuon, A. Egier, and J. Rose, "Design, layout and verification of an FPGA using automated tools," in *FPGA 2005*, H. Schmit and S. J. E. Wilton, Eds. ACM, 2005, pp. 215–226. [Online]. Available: http://doi.acm.org/10.1145/1046192.1046220
7. N. Voros, A. Rosti, and M. Hübner, Eds., *Dynamic System Reconfiguration in Heterogeneous Platforms*, ser. Lecture notes in electrical engineering. Springer Netherlands,, 2009., vol. 40.
8. N. Kafafi, K. Bozman, and S. J. Wilton, "Architectures and algorithms for synthesizable embedded programmable logic cores," in *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*. ACM, 2003, pp. 3–11.
9. V. A. Ova, G. Lemieux, and R. Saleh, "An improved "soft" efpga design and implementation strategy," in *Proceedings of the IEEE 2005 Custom Integrated Circuits Conference, CICC 2005.* IEEE, 2005, pp. 179–182. [Online]. Available: http://dx.doi.org/10.1109/CICC.2005.1568636
10. V. Betz and J. Rose, *Field-Programmable Logic and Applications: 7th International Workshop, FPL '97 London, UK, September 1–3, 1997 Proceedings*. Springer Berlin Heidelberg, 1997, ch. VPR: a new packing, placement and routing tool for FPGA research, pp. 213–222.
11. T. Wiersema, A. Bockhorn, and M. Platzner, "Embedding fpga overlays into configurable systems-on-chip: Reconos meets zuma," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–6.
12. C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, pp. 5–35, 1991.
13. U. of California Berkeley. (1992) Berkeley logic interchange format(blif). [Online]. Available: http://vlsi.colorado.edu/~vis/blif.ps
14. P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin 2 - an open-source verilog hdl synthesis tool for cad research," in *FCCM 2010*, 2010.
15. R. Brayton and A. Mishchenko, *Computer Aided Verification: 22nd International Conference, CAV 2010*. Springer Berlin Heidelberg, 2010, ch. ABC: An Academic Industrial-Strength Verification Tool, pp. 24–40.
16. W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of Applied Physics*, vol. 19, no. 1, pp. 55–63, 1948.
17. R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *FCCM*, ser. FCCM '01. IEEE Computer Society, 2001, pp. 227–238.